Purpose: The purpose of this project was to implement a fast and efficient memory allocator that can allocate, free, and reallocate memory space in the heap using concepts from lecture such as the binned free list (BFL) approach. The performance of the allocator was tested on various traces against the standard C memory allocator for memory utilization (M) and throughput (T) where M is the ratio of the memory requested over the total amount of memory allocated and T is the amount of operations per second

Overview: We first developed a simple BFL for blocks of size 2^i. Due to poor memory utilization from internal fragmentation (achieving M ~53% and T ~55%) we moved to a variation of BFL where each bin holds a range of blocks from $[2^i,2^{i+1})$ leading to M ~71% and T ~55%. We then implemented bidirectional coalescing and a more efficient reallocator giving us M ~82% and T ~ 99%. Freed blocks are stored to the BFL as a unique `struct free_block_t` that would contain first a pointer to the next `free_block_t` in our original implementation to in our coalescing implementation with both a pointer to the next and previous `free_block_t` to make adding and removing blocks from the BFL occur in O(1) time when coalescing.

Initial BFL Implementation: Our initial BFL used bins of blocks sized 2^i starting from 16 bytes (enough for a minimum request and our header). When a new block is requested, if a larger bin holds a block, it would be split into two smaller bins recursively until the bin containing the smallest/most efficient size would guarantee enough memory allocated. If no larger bin is nonempty, we request two blocks of the requested size to minimize the direct requests, using `mem_sbrk`, where one is returned while the other is added on to the binned free lists.

The blocks are sized 2^4 to 2^{31} since we used the first 8 bytes of each block to store each block's size, so when pointers were freed, we would know where to place the free block by simply inspecting the 8 bytes of memory preceding the pointer returned to the user. Furthermore, since all data was 8 byte aligned, we would need 16 bytes for even the smallest request, in order to maintain the header appropriately.

This implementation ran with M ~53% and T ~55%. The low memory utilization we determined was likely due to the internal fragmentation in our 2^i sized blocks, a known downside of BFL. The remaining space in a block was impossible to utilize until it was freed and a size closer to the size of the block was requested, an unlikely and very conditional event. The low speed was solely due to a poor reallocator as traces that with reallocation slowed to T ~0-1%.

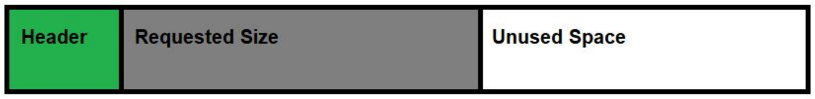

Figure 1. Should a memory size request be just greater than a power of 2, you waste almost half the internal memory allocated.

Minimum Block Size: To remember the size of our block when it is eventually freed, we need to store it with the block which we did with a header in front of the block. When the block is freed we do not need to consider the contents of the memory space it represented and as such we can cast it to our `free_block_t struct` to take advantage of the now unused space. This allows us to safely store both the size of the block and a pointer to the next `free_block_t` in a singly linked list of free blocks. When we did not have any notion of coalescing we could use the first version depicted in Figure 3 of our minimum block (8 bytes for header and 8 blocks for our `struct`). However, when coalescing with blocks around it in memory space, we do not know where in our BFL could the other blocks be nor if those blocks are currently in use. As such we needed to add a footer to represent if a block was in use or not and our `free_block_t struct` now required another pointer to the previous `free_block_t` in order to avoid searching through all blocks in our BFL and instead just link the neighboring `free_block_t` to each other in O(1) time, leading to a minimum block size of 32 bytes.

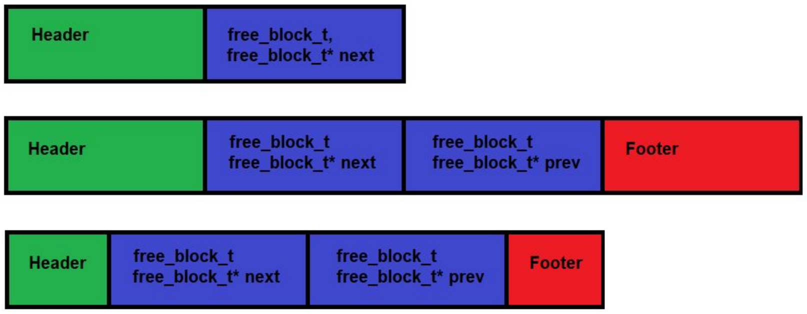
Header
free_block_t, free_block_t* next
Header
free_block_t free_block_t* next
free_block_t free_block_t* prev
Footer
Header
free_block_t free_block_t* next
free_block_t free_block_t* prev
Footer

Figure 2. Large blocks are 8 bytes and small blocks are 4. We go from a minimum block of 16 bytes without coalescing, to 32 with coalescing, and 24 with more efficient headers and footers.

Because we know that we only consider allocations of max size 2^{31}-1, we can in fact store the size of the request in a `uint32_t` instead of a `size_t`, reducing our minimum block size to 24 bytes. However as we see from the last minimum block size in Figure 3, the data is no longer 8 byte aligned. To compensate at the very start of our memory allocator we allocate 4 dummy bytes to ensure 8 byte alignment.

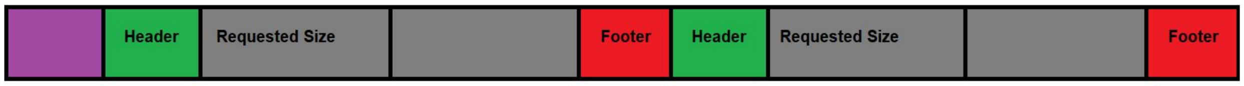

Figure 3. Four bytes are allocated at initialization so the actual memory space is 8 byte aligned.

Using Exact Block Sizes: The presence of immense internal fragmentation led us to implement an allocator that would allocate almost exactly the correct amount of memory requested (rounded up to the nearest multiple of 8 for data alignment purposes). This new implementation uses a modified BFL algorithm which would instead store ranges of blocks where the ith-bin holds blocks in the range [2^{i+4},2^{i+5}) where bins whose minimum block size would satisfy the requested size would be used. To further prevent internal fragmentation, when a block is found, we split a block's excess memory space if the remaining unused portion can support a different size request. This led us to M ~65 % and T ~55%.

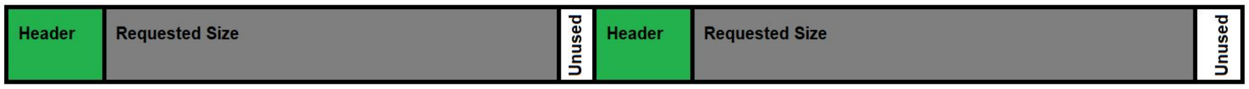

Figure 4. With fitting our block as close as possible to the requested memory size, our blocks guarantee internal fragmentation less than 8 bytes.

Coalescing: In order to prevent as much external fragmentation as possible we implemented bidirectional coalescing which required us to know whether the neighboring blocks in memory were in use or not as well as their size, now requiring a footer to look at blocks upstream in memory. In the footer we stored either the size of the block if it was free or a tag which we set to 1 (since a block can never be size 1) when in use. If a block is free, we can then use the recorded size in the footer to get the pointer upstream of it. Knowing the size of the pointer, we can also access the pointer downstream and access its footer to know if it is in use or not. Due to our varied block range BFL implementation, we can simply concatenate the two blocks, adjust the footer and header and store the coalesced free block in the appropriately sized bin. This led us to M ~ 71% and T ~ 55%.

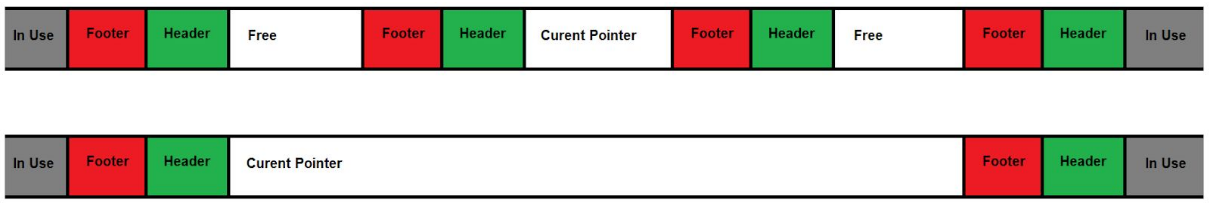

Figure 5. We execute bidirectional coalescing when the current pointer is being freed and coalesce until the neighboring blocks are in use, returning the pointer at the start of the block.

Improving Realloc: After implementing the use of exact block sizes, we observed that our memory allocator was extremely slow on traces that had reallocation. We covered cases where the size requested was less than or equal to the current size of the pointer by using the same pointer given. However, we realized that it was common for the pointer that is being reallocated to be at the end of the heap and could call `mem_sbrk` to extend and modify the size of the given pointer instead of allocating an entire new block. This led us to M ~82% and T ~99%.

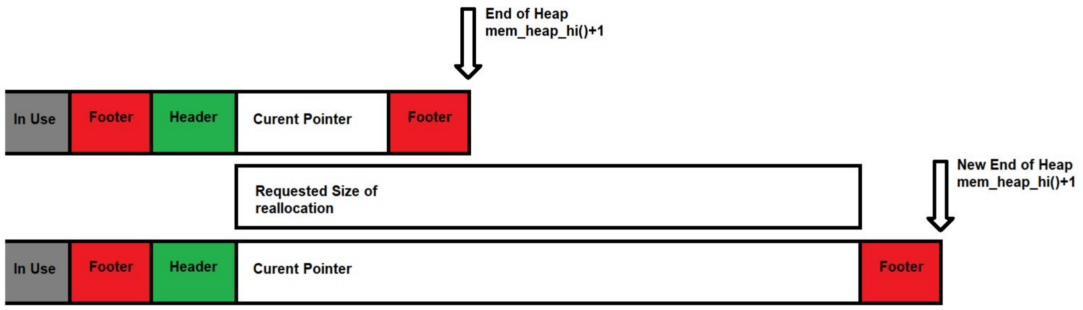

Figure 6. If the current pointer is at the end of the heap and we need a bigger block, we can directly extend the heap than searching for and allocating a new block and freeing the previous.

Validator: We ensure our implementation of the allocator is correct by ensuring that an operation does not violate any of the validator's invariant checks. The validator was calibrated by ensuring that no errors occurred with the naive correct implementation and that when run with the bad allocator, produced the correct errors. Our validator checks whether the current load/size request violates any of our invariants namely: allocated ranges returned by the allocator must be 8 byte aligned, allocated ranges returned by the allocator must be within the heap, and allocated ranges returned by the allocator must not overlap with other allocation.

Lastly for reallocation, we must ensure the original data of the allocation is intact up to the reallocated size. We do this by creating a unique tag byte that fills an allocation based on a random value seeded by the index of that allocation added to said index. This ensures a unique value in the contents of the allocation that we can quickly and efficiently check for after an allocation. For the reallocation to be correct, each byte in the new allocation should be correct up to the minimum of the size of the original allocation and the reallocation size.

Team dynamics: For the initial BFL implementation we came up with all the necessary helper functions and divided the work evenly. After the first pass, Miller was more focused on introducing new optimizations into the code and future proofing any necessary parts while Allen was focused on simplifying and improving on implemented ideas while also ensuring our correctness invariant was maintained for any new introductions. The project could not have been completed to a satisfiable degree without one or the other.

Post Mortem Beta: We were satisfied with our performance for the beta. The initial pass at a BFL as well as the binned ranges was simple and relatively effective. Introducing proper coalescing however was extremely difficult to get correct as the pointer math became very confusing. We tried to as effectively as possible black box this with helper functions. We ended up not using the OpenTuner because we wanted our beta to be as general as possible for any possible trace. Given the context of other implementations for the final we will likely tweak using the OpenTuner .

Further Improvements: While our reallocation works well for known traces that use reallocation, we believe there are cases for it to be improved such as coalescing with the blocks around the pointer or cutting off the excess space of the pointer into a new block when the requested size is smaller. We will also use the OpenTuner to a greater degree to find the optimal settings of fixed variables (minimum block size, coalescing frequency, etc.). We also believe there can be optimizations to our malloc and free since traces that solely used those sometimes had very poor utilization. Some initial ideas that come to mind are a larger threshold for a minimum block that can be split off (opentuner can discover this) or a free that doesn't coalesce as much as possible (seemingly unintuitive but possibly effective).

Project Log:

#	Date	Start time	Duration		Description
			Miller	Allen	
1	10/21/2020	7:30 PM	1	1	Initial team meeting, writing contract, planning future meets
2	10/22/2020	7:00 PM	1	1	Pair ideation, dividing helper function work
3	10/23/2020	6:00 PM	2		Implemented helper functions
4	10/24/2020	5:00 PM		2	Implemented helper functions for BFL and validator
5	10/25/2020	11:00 AM	5	4	Complete simple BFL implementation
6	10/26/2020	6:00 PM	2		Groundwork for coalescing
7	10/27/2020	4:00 PM	6	4	Coalescing functioning, fleshing out valdator
8	10/28/2020	12:00 AM		2	Faster reallocation, fixing validator
9	10/28/2020	11:00 PM	1		Refactoring and simplifying functions
10	10/29/2020	10:00 AM	2	2	Completed beta report
Total			20	16	